FCOM COM & ActiveX interface

by Thomas Dixon

Thomas Dixon

Published by Microprocessor Engineering

FCOM COM & ActiveX interface
User manual
Manual revision 1.0
2 October 2008

Software
Software version 1.0

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 COM Interface Library

FCOM is an interfacing library to COM objects.

The interfacing is made easier by use of type libraries, which contain descriptions of COM objects and what structures and the globals they use.

The FCOM library is loaded into its own vocabulary to reduce the chance of redefining words. When the COM interface needs to be used include the line `also FCOM` in the source before using the words in the library.

The external API interface used in this library diverts from normal MPE practice by defining the API words as `"PASCAL"`. As a result, the Forth stack order for arguments is reversed from that in the MSDN documentation. This done to maintain portability of code across various Forth implementations.

The following code compiles the FCOM library.

```
include FCOMbase.bld
```

which includes a harness file, the Unicode and GUID code, plus the FCOM file itself. If your Forth is not supported yet, you will have to use one of the existing harnesses, e.g. *VfxHarness.fth* as a base for your own.

For a very simple example on how to use the FCOM interface library, please look at the *SAPI.fth* text to speech example. Also provided are *MSCAL.fth* to put an ActiveX control in a child window, and *Flash.fth*, which plays a Flash movie in a child window.

1.1 Type Library Management

The following words are used to attach and detach type libraries from the programming environment. Type libraries contain quite a bit of information helpful in interfacing to COM objects.

`: typelib ( major minor <guid> -- )`

This attaches a registered type library. The type library must exist in the windows registry in order for it to succeed. The major and minor arguments are integers specifying the typelibrary version. Once the type library has been attached, it will stay attached until removed with free-lasttypelib or freetypelibs (or the programming environment is closed.

`: typelibfile ( str len -- )`

This attaches a type library found in a file. The file may be a standalone type library (as compiled with the MIDL compiler) or a DLL or EXE that has the type library as a resource.

`: free-lasttypelib ( -- )`

This will free the last type library that was attached to the programming environment.

`: freetypelibs ( -- )`

This will free all attached type libraries.

1.2 Initialization

This section details some of the initialization and cleanup used in COM.

The most important thing to remember in COM development is that all components are garbage collected using reference counting. Querying an interface will increment the reference cound as will the `IAddRef` method. Decrementing the reference is done through the `IReleaseRef`* method. Once the reference count on all interfaces to an object have returned to zero, the object is removed from memory. Properly using these methods will avoid memory leaks.

`: com_init ( -- )`
This initializes the process with the `CoInitialize` Windows function. It must be called in turnkey applications before trying to create COM objects. It is, by default, included in the `atCold` initialization chain, so most applications don't need to use it (but some will).

1.3 Interface Definitions

Interfaces don't have to be attached in type libraries in order to use them. The words in this section are used to define interfaces directly in source. This is useful when the typelibrary does not exist, but header files are provided for a particular component.

`: Interface ( interface <guid> -- )`
Defines a new interface and its corresponding GUID.

`: Open-Interface ( interface -- )`
Opens an interface for defining methods.

`: Close-Interface ( -- )`
Closes the currenly-open interface so that methods are no lonter assigned to that interface when declared.

`: IMethod ( n i -- )`
Defines a new method in the virtual table of an interface. *N* is the number of arguments, and *i* is the vector index in the vtable.

1.4 Exploring Type Libraries

These words are helpful for exploring the "metadata" contained in type libraries. Type libraries can contain interfaces, coclasses, structures, and constants that are used with particular components. Type libraries are usually found in the windows registry (when components are regsitered), but may also be resources in a dll or executable.

`: Interfaces ( -- )`
Lists all the interfaces currenly accessible in the attached type libraries.

`: CoClasses ( -- )`
Lists all the coclasses currently accessible in the attached type libraries. A coclass is basically the GUID for a component.

`: ComConsts ( -- )`
Lists all the enumerations currently accessible in the attached type libraries. You will need to execute the Enumeration name to see what constants are part of the enumeration.

`: Structures ( -- )`
Lists all the COM-based structures currently accessible in the attached type libraries.

1.4.1 Exploring Interfaces

Interfaces in an attached type library can be explored by typing the interface name followed by `words`. This will list all the interface methods. It should also list all the arguments required by each method in the proper order. Below is an example that will list out all the methods in the `ISpVoice` interface - the SAPI type library must be attached in order for the example to work).

```
ISpVoice words
>>> will return:
  2 3   IMethod SetNotifySink ( *ISpNotifySink -- hres )
  5 4   IMethod SetNotifyWindowMessage ( LONG_PTR UINT_PTR n wireHWND -- hres )
  4 5   IMethod SetNotifyCallbackFunction ( LONG_PTR UINT_PTR **void -- hres )
  4 6   IMethod SetNotifyCallbackInterface ( LONG_PTR UINT_PTR **void -- hres )
  1 7   IMethod SetNotifyWin32Event ( -- hres )
  2 8   IMethod WaitForNotifyEvent ( n -- hres )
  1 9   IMethod GetNotifyEventHandle ( -- ptr )
  5 10  IMethod SetInterest ( d d -- hres )
  4 11  IMethod GetEvents ( *n *SPEVENT n -- hres )
  2 12  IMethod GetInfo ( *SPEVENTSOURCEINFO -- hres )
  3 13  IMethod SetOutput ( n IUnknown -- hres )
  2 14  IMethod GetOutputObjectToken ( **ISpObjectToken -- hres )
  2 15  IMethod GetOutputStream ( **ISpStreamFormat -- hres )
  1 16  IMethod Pause ( -- hres )
  1 17  IMethod Resume ( -- hres )
  2 18  IMethod SetVoice ( *ISpObjectToken -- hres )
  2 19  IMethod GetVoice ( **ISpObjectToken -- hres )
  4 20  IMethod Speak ( *n n lpwstr -- hres )
  4 21  IMethod SpeakStream ( *n n *IStream -- hres )
  3 22  IMethod GetStatus ( *lpwstr *SPVOICESTATUS -- hres )
  4 23  IMethod Skip ( *n n lpwstr -- hres )
  2 24  IMethod SetPriority ( SPVPRIORITY -- hres )
  2 25  IMethod GetPriority ( *SPVPRIORITY -- hres )
  2 26  IMethod SetAlertBoundary ( SPEVENTENUM -- hres )
  2 27  IMethod GetAlertBoundary ( *SPEVENTENUM -- hres )
  2 28  IMethod SetRate ( n -- hres )
  2 29  IMethod GetRate ( *n -- hres )
  2 30  IMethod SetVolume ( h -- hres )
  2 31  IMethod GetVolume ( *h -- hres )
  2 32  IMethod WaitUntilDone ( n -- hres )
  2 33  IMethod SetSyncSpeakTimeout ( n -- hres )
  2 34  IMethod GetSyncSpeakTimeout ( *n -- hres )
  1 35  IMethod SpeakCompleteEvent ( -- ptr )
  5 36  IMethod IsUISupported ( *n n *void *h -- hres )
  6 37  IMethod DisplayUI ( n *void *h *h wireHWND -- hres )
```

1.4.2 Exploring Structures

Structures in an attached type library can be explored by typing the structure name followed by "words". Lists out all the fields in the structure. It should also add a comment as to what type is expected in the structure. The example below shows a structure used in the SAPI type library

```
SPVOICESTATUS words
>>> will return:
  0  Field: ulCurrentStream \ (n)
  4  Field: ulLastStreamQueued \ (n)
  8  Field: hrLastResult \ (hres)
  12 Field: dwRunningState \ (n)
  16 Field: ulInputWordPos \ (n)
  20 Field: ulInputWordLen \ (n)
  24 Field: ulInputSentPos \ (n)
  28 Field: ulInputSentLen \ (n)
  32 Field: lBookmarkId \ (n)
  36 Field: PhonemeId \ (h)
  40 Field: VisemeId \ (SPVISEMES)
  44 Field: dwReserved1 \ (n)
  48 Field: dwReserved2 \ (n)
```

1.4.3 Exploring Enumerations

All constants in a type library are categorized in enumerations. In order to see what constants are in an enumeration, type the enumeration name, as shown in the example below.

```
SpeechVoiceSpeakFlags
>>> will return:
SVSFDefault
SVSFlagsAsync
SVSFPurgeBeforeSpeak
SVSFIsFilename
SVSFIsXML
SVSFIsNotXML
SVSFPersistXML
SVSFNLPSpeakPunc
SVSFNLPMask
SVSFVoiceMask
SVSFUnusedFlags
```

1.5 Interface and Structure Instantiation

These words are for instatiating interfaces or COM-based structures.

`: ComIFace ( interface -- )`
Defines a interface based on a interface definition. The interface definition may be declared in the source file, or located in an attached type library.
Once an interface has been instantiated, methods are called by following the instance name with the method to call. If the word following the instance is not a method, the instance will just return a pointer to itself.

`: Struct ( structtype -- )`
Defines a structure based on a COM structure definition (structtype). The structtype can be declared in the source file, or located in an attached type library. Once a structure has been instantiated, a field address can be calculated by following the instance name with the field name. If no valid field name is found, the address to the structure itself is returned. If the structure has nested structures, the structure name will have the same behavior as the instance - so nested structures may have several field names following the instance name.

`: COM ( addr <name> <method> --  )`
This takes any pointer and "casts" it to a given interface and executes a method associated with that interface. The interface name may be declared in source or in an attached typelibrary. The behavior should be identical to executing an instance name.

`: USEStruct ( addr <structtype> -- addr )`
This takes any pointer and "casts" it to a given COM structure. The structure may be declared in source or in an attached type library. The behavior should be identical to executing an instance name.

`: FreeRef ( iface -- )`
Release the interface.

1.6 Variant Types

COM interfaces will sometimes have variant types as arguments. Variants are heavily used in dispatch interfaces as well. A variant is a structure that is meant to be flexible enough to hold many different types in a single structure. They also are self describing, so they will indicate what kind of typed data is within them. The following lists possible variant types (which are constants in the library).

- 0 = VT_EMPTY \ no data associated with this
- 1 = VT_NULL \ same as a regular NULL
- 16 = VT_I1 \ 1 byte signed integer
- 17 = VT_UI1 \ 1 byte unsigned integer
- 2 = VT_I2 \ 2 bytes signed integer
- 18 = VT_UI2 \ 2 bytes unsigned integer
- 3 = VT_I4 \ 4 bytes signed integer
- 22 = VT_INT \ same as VT_I4 but with a different code
- 19 = VT_UI4 \ 4 bytes unsigned integer
- 23 = VT_UINT \ same as VT_UI4 but with a different code
- 20 = VT_I8 \ 8 bytes signed integer
- 21 = VT_UI8 \ 8 bytes unsigned integer
- 4 = VT_R4 \ IEEE 32-bit floating-point number
- 5 = VT_R8 \ IEEE 64-bit floating-point number
- 6 = VT_CY \ 8 byte two's complement integer (scaled by 10000, used for currency)
- 7 = VT_DATE \ 64-bit floating-point number representing the days since Dec. 31, 1899
- 8 = VT_BSTR \ pointer to Null-terminated unicode string–see unicode notes above
- 9 = VT_DISPATCH \ pointer to a IDispatch interface
- 11 = VT_BOOL \ boolean value
- 10 = VT_ERROR \ 32-bit number containing status code
- 13 = VT_UNKNOWN \ pointer to a IUnknown interface
- 64 = VT_FILETIME \ 64-bit FileTime structure (see Win32 API)
- 30 = VT_LPSTR \ pointer to Null-terminated ANSI string
- 31 = VT_LPWSTR \ pointer to Null-terminated Unicode string
- 72 = VT_CLSID \ pointer to UUID (or CLSID, or GUID)
- 71 = VT_CF \ pointer to a clip structure
- 65 = VT_BLOB \ 32-bit count of bytes followed by that number of bytes

- 70 = VT_BLOBOBJECT \ a blob containing a serialize object
- 66 = VT_STREAM \ pointer to an IStream interface
- 68 = VT_STREAMED_OBJECT \ same as stream, but contains an object
- 67 = VT_STORAGE \ pointer to an IStorage interface
- 69 = VT_STORED_OBJECT \ same as storage, but contains an object
- 14 = VT_DECIMAL \ a decimal structure
- 24 = VT_VOID \ void
- 25 = VT_HRESULT \ standard return code
- 26 = VT_PTR \ pointer to something
- 27 = VT_SAFEARRAY \ safearray
- 28 = VT_CARRAY \ normal array type
- 29 = VT_USERDEFINED \ user defined type
- $1000 = VT_VECTOR \ array of types, pointer to count, then pointer to the array
- $2000 = VT_ARRAY \ pointer to safearray
- $4000 = VT_BYREF \ value is a reference
- $8000 = VT_RESERVED \ reserved type
- $FFFF = VT_ILLEGAL \ illegal variant
- 12 = VT_VARIANT \ type indicator followed by corresponding value
- $FFF = VT_ILLEGALMASK \ Illegal variant mask.
- $FFF = VT_TYPEMASK \ used as a mask for vt_vector, array and what-not

Another thing to note with variants is that when calling a virtual table method that specifies a variant as an object, it will expect that the whole variant is on the stack and not just a pointer. This takes up four cells on the stack to represent one variant. The order that data should be put on the stack is with the variant on the top of stack, and the data starting at the third to top of the stack. The example below shows how an integer (123) would be placed on the stack as a variant

```
0 123 0 VT_INT
>>> method using a variant can then be called
```

`: vt>Str ( vt -- str len )`
This will convert a given VT constant to a readable type

1.7 Dispatch Interfaces

Dispatch interfaces are horribly ugly creatures designed for use in dynamic scripted languages. They perform runtime type checking and late-late binding (dispatching on a dispatched interface). They are not fast, but very flexible in taking all kinds of input arguments to "get the job done". Because of the increased complexity, dispatch interfaces are treated quite differently to normal vtable interfaces.

Probably the biggest difference in how dispatch tables are called is by the use of a variant stack. Arguments used in a dispatch interface call are pushed onto the variant stack beforehand. This allows the dispatch interface to do its runtime type checking and such.

`: Dispatcher ( <name> <progID> -- )`

This will define a dispatched interface based on a given program ID (same program id's used in visual basic). The defined word will call methods in the same fashion as virtual interfaces, but on a late-bound dispatched basis. This interface will need to be released when usage is done (just like all other interfaces).

`: DispLate" ( interface "<method>" -- hres )`
This does a late-late-bound dispatch binding (for the truly depraved). It is used on a dispatch interface. The method is called as a string following the word similar to string processing words such as `S"`. This kind of method dispatch is required when the exact component is not known beforehand.

1.7.1 Variant Stack

The variant stack acts like a separate stack that arguments are pushed onto before calling a dispatched method. After a method has been called, the return value is stored in a separate location and is accessed through `retVT@`.

`: >VT ( n VT -- )`
Pushes a data value onto the variant stack. VT is the variant type. Note that for some types, two data stack items are pushed, e.g. for double length floating point and 64-bit numbers.

`: VT> ( -- n VT )`
Pops a value from the variant stack onto the data stack. VT is the variant type. Note that for some types, two data stack items are popped onto the data stack.

`: .vt ( -- )`
Prints the variant stack data (similar to the `.s` word).

`: retVT@ ( -- n VT )`
This returns that return value of the last dispatch method invocation. VT is the variant type of the return value.

2 Globally Unique IDentification (GUID) Library

This library consists of a few words that are for defining GUIDs. GUIDs are used mostly by Foreign Function Interfaces such as COM. In essence, they are just 128-bit identifiers, but usually are expressed in a format of {00000000-0000-0000-0000-000000000000}.

`: Guid, ( -- )`
Parses the input stream for a GUID and inserts it into the dictionary.

`: CLSID>Str ( addr -- str len )`
Converts a CLSID (or GUID) to a string

`: UUID ( <name> "guid" -- )`
Defines a UUID, e.g.

```
Example: UUID test {01234567-89AB-CDEF-0000-000123456789}
```

3 Unicode String Library

This provides basic words for working with Unicode strings. These Unicode strings are appended with a 16 bit null character. This is because of the various abominations in the Microsoft documentation as to whether this is needed or not.

`: UniPlace ( addr len destaddr -- )`
Store a unicode string to an address

`: UniAppend ( addr len destaddr -- )`
Append a string to the end of an address

`Synonym +UniPlace UniAppend`
Same as UniAppend

`: UniCount ( addr -- addr len )`
Fetch a unicode string from an address (stored with uniplace)

`: ZUniCount ( addr -- addr len )`
Fetch a null-terminated unicode string from an address (null is 16-bit)

`: UniType ( addr len -- )`
Type a unicode string to the console

`: Ustr, ( addr n -- )`
Add a unicode string to the dictionary at HERE.

`: Asc>Uni ( str len -- str len )`
Convert an ASCII string to Unicode. You **must** free the string with `FREE` when it is no longer needed.

`: Uni>Asc ( str len -- str len )`
Convert a Unicode string to ASCII. You **must** free the string with `FREE` when it is no longer needed.

`: >Unicode ( str len -- str len )`
Convert an ASCII string to Unicode. The returned string is in `PAD`.

`: U16>ascii ( str len -- str len )`
Convert a Unicode string to ASCII at `PAD`.

`: U" ( ... " -- str len )`
Unicode string - should behave the same way as s"

3.1 BSTR words

Some APIs require more specific conditions to their unicode strings. (ie: distributed and network apis) bstr has more constraints applied to it. These words are to convert to bstrs and back again.

`: Asc>bstr ( str len -- bstr )`
Convert ascii string to unicode bstr. bstr must be freed later with 'bstrfree'.

`: bstrFree ( bstr -- )`
Free a BSTR.

`: bstrlen ( bstr -- len )`
Returns the length of the bstr. From this the bstr can be used with all the other unicode functions.

4 Excel Interface Library

This library contains a number of words to exchange data with Excel.

`: Excel-close ( -- )`
Closes the Excel Interface and de-allocates all objects.

`: Excel-init ( -- )`
Initializes Excel and establishes interface. Excel is initialized in a separate process.

`: Visible! ( f -- )`
Sets the visibility of Excel. Calling this word with 1 will make Excel visible, 0 will hide it (default behavior).

`: Excel-new ( -- )`
Creates a new workbook in Excel

`: Excel-Open ( str len -- )`
Opens an Excel file

`: Excel-Save ( -- )`
Saves the current Excel file.

`: Excel-SaveAs ( str len -- )`
Saves the current Excel file under a given file name.

`: XLf@ ( x y -- float )`
Fetches the floating point value in cell (x,y) on the current worksheet. If the value is not a floating point, an exception will be thrown.

`: XL@ ( x y -- n )`
Fetches an integer value in cell (x,y) on the current worksheet. If the value is not a number, an exception will be thrown.

`: XLstr@ ( x y -- str len )`
Fetches the value in cell (x,y) as a string.

`: XLstr! ( str len x y -- )`
Stores a string to cell (x,y) in the current worksheet. The string may be text, but can also be an Excel formula.

`: XL! ( n x y -- )`
Stores an integer to cell (x,y) in the current worksheet.

`: XLf! ( float x y -- )`
Stores a floating point number to cell (x,y) in the current worksheet.

5 ActiveX Control Utilities

These words are used for hosting ActiveX controls in child windows. It takes care of most of the container interfacing so that the ActiveX control can be treated as a child window.

`: AXCreate ( str len hwnd -- )`
Creates an ActiveX control and attaches it to a window handle.

`: AXGetIUnknown ( addr hwnd -- )`
Stores the IUnknown of an ActiveX object in the hwnd window to addr.

`: AXQuery ( addr IID hwnd -- )`
Queries a given interface to an ActiveX control (in a windows handle) and stores the interface in addr.

6 VFX Forth harness to FCOM

The file *VfxHarness.fth* provides the Forth system dependencies between VFX Forth for Windows and Tomas Dixons's FCOM library. Other files called *xxxHarness.fth* provide the dependencies for other systems.

There are three areas that require changes between Forth systems.

- Error handling.
- How the Windows API is defined.
- How COM calls are made.

6.1 Error Handling

In order to ease error handling in complex applications, fatal errors cause a `THROW`. Associated with each error is a constant and (perhaps) a string. In VFX Forth, this association is provided by `ErrDef`, which creates a constant using the next error number and associates a string with it, e.g.

```
ErrDef err_AXcreate  "Unable to create ActiveX Control"
```

The word `.THROW ( n -- )` can then be used to display the corresponding string.

```
: ?THROW        \ flag throw-code --
```

Perform a `THROW` of value *throw-code* if *flag* is non-zero.

```
ErrDef err_NotDispatch  " Dispatch Interface not returned"
ErrDef err_InvBind      " Invalid Bind Type"
ErrDef err_Bind         " Unable to Bind to type"
ErrDef err_GetTypeInfo  " Unable to get type info"
ErrDef err_GetDoc       " Unable to get Documentation"
ErrDef err_LoadTypeLib  " Error Loading Type Library"
ErrDef err_GetTypeComp  " Error Getting TypeComp"
ErrDef err_NoVI         " No Virtual Interface"
ErrDef err_NotIF        " Not An Interface"
ErrDef err_VarStFull    " Variant Stack Full"
ErrDef err_CantDispatch " Unable to Call Dispatch"
ErrDef err_NoProgID     " Unable to Find ProgID"
ErrDef err_NoIUnknown   " Unable to Get IUnknown"
ErrDef err_NoValStruct  " Not a valid COM Structure"

ErrDef err_BadAlloc     " Unable to Allocate String"

ErrDef err_GUIDlen      " Invalid Guid Length"
ErrDef err_BadCLSID     " Not a CLSID"

ErrDef err_IUnknown     " Unable to get IUnknown interface"
ErrDef err_AXcreate     " Unable to create ActiveX Control"
ErrDef err_noIF         " Unable to get Interface"
```

6.2 Windows API access

The FCOM library accesses API calls with the Forth stack order reversed from that of the MSDN documentation, i.e. if MSDN indicates:

```
int foo( int a, int b,int c );
```

the Forth stack picture for `foo` will be:

```
c b a -- int
```

IN other words, the left-most argument in the MSDN documentation must be on top of the stack on entry.

To avoid name collisions and breaking existing code, FCOM accesses words with the same name as the API call, but preceded by 'f', e.g. the `CoCreateInstance` API call is called through the Forth word `fCoCreateInstance`. This arrangment permits different Forth systems to insulate their particular standards and conventions from the FCOM library.

6.2.1 GUIDs

`: fStringFromCLSID      \ ppsz rclsid -- res`
Direct call to `StringFromCLSID()` API call.

6.2.2 UniCode functions

`: fMultiByteToWideChar  \ wclen *wc mblen *mb flags cp -- int`
Direct call to `MultiByteToWideChar()` API call.

`: MB>WC         \ wclen *wc mblen *mb flags cp -- int`
Convert MultiByte string to 16 bit Unicode.

`: WC>MB         \ *used *def mblen *mb wclen *wc flags cp -- len`
A direct call to the `WideCharToMultiByte()` API function.

`: fSysAllocStringLen    \ cch pch -- addr`
A direct call to the `SysAllocStringLen()` API function.

`AliasedExtern: fSysFreeString void "PASCAL" SysFreeString( void * bstr );`
A direct call to the `SysFreeString()` API function.

6.2.3 COM primitives

`: fCoInitialize \ NULL -- hresult`
A direct call to the `CoInitialize()` API function.

`: fCoCreateInstance     \ ppv riid context pUnkOuter rclsid -- res`
A direct call to the `CoCreateInstance()` API function.

`: fCLSIDFromProgID      \ pclsid pprogid -- res`
A direct call to the `CLSIDFromProgID()` API function.

`: fLoadRegTypeLib       \ pptlib lcid wverminor wvermajor rguid -- res`
A direct call to the `CLSIDFromProgID()` API function.

`: fLoadTypeLib  \ pptlib szFile -- res`
A direct call to the `LoadTypeLib()` API function.

`AliasedExtern: fCoTaskMemFree void "PASCAL" CoTaskMemFree( *pv );`
A direct call to the `CoTaskMemFree()` API function.

6.2.4 ActiveX

`: fAtlAxCreateControl   \ ppUnk pstream hWnd lpszName -- res`
A direct call to the `AtlAxGetControl()` API function.

`: fAtlAxGetControl      \ pptlib szFile -- res`
A direct call to the `AtlAxGetControl()` API function.

6.3 COM calls

`: INTERFACE-CALL ( n1 n2 n3 ... nx a indx )`
Vtable call.

`: RUN-INTERFACE ( pointer imethod -- )`
Run-time interface call.

`: COMPILE-INTERFACE ( pointer imethod -- )`
Fast compile interface call

`: RUN-VTABLE ( pointer offset -- )`
Vtable call.

`: COMPILE-VTABLE ( pointer offset -- )`
Compile a call through the Vtable.

6.4 Miscellaneous

`: laddr ( "<name>" -- addr )`
At run-time, returns the address of the local variable.

`defer COMsearcher ( str len -- flag )`
This is a `DEFER`red word used to avoid a forward reference.

`:noname 2drop false ; is COMsearcher`
The default action until `COMsearcher` is set.

`: (comfind) ( str len -- flag )`
A search primitive that allows type libraries to be searched. Depending on the host system it is either hooked into *\fo{FIND) or hooked into the action when a token is undefined.

`: str,          \ caddr len --`
Lay the string into the dictionary at `HERE`, reserve space for it and `ALIGN` the dictionary.

Index

S

T

U

V

W

X

Z